Read-Log-Update
A Lightweight Synchronization Mechanism for Concurrent Programming

Paper Reading Group

Alexander Matveev
Nir Shavit
Pascal Felber
Patrick Marlier
Presents: Maksym Planeta

24.09.2015

Table of Contents

Table of Contents

Motivation

What is bad with LRU?

- Complex to use for a writer;
- Optimized for **low** number of writers
- High delays in `synchronize_rcu`

Contributions

RCU + STM = RLU.

- Update several objects with single counter increment;

Contributions

RCU + STM = RLU.

- Update several objects with single counter increment;
  Traverse doubly linked lists in both directions!

Contributions

RCU + STM = RLU.

- Update several objects with single counter increment; Traverse doubly linked lists in both directions!
- Stay compatible with RCU

RCU recap

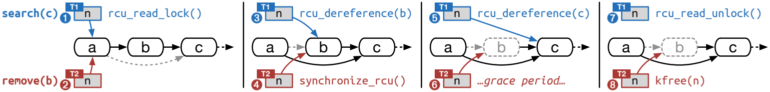


Figure 2. Concurrent search and removal with the RCU-based linked list.

Single point manipulation

```
static inline void
__list_add_rcu(struct list_head *new,
               struct list_head *prev,
               struct list_head *next)
{
    new->next = next;
    new->prev = prev;
    rcu_assign_pointer(list_next_rcu(prev), new);
    next->prev = new;
}
```

RLU style

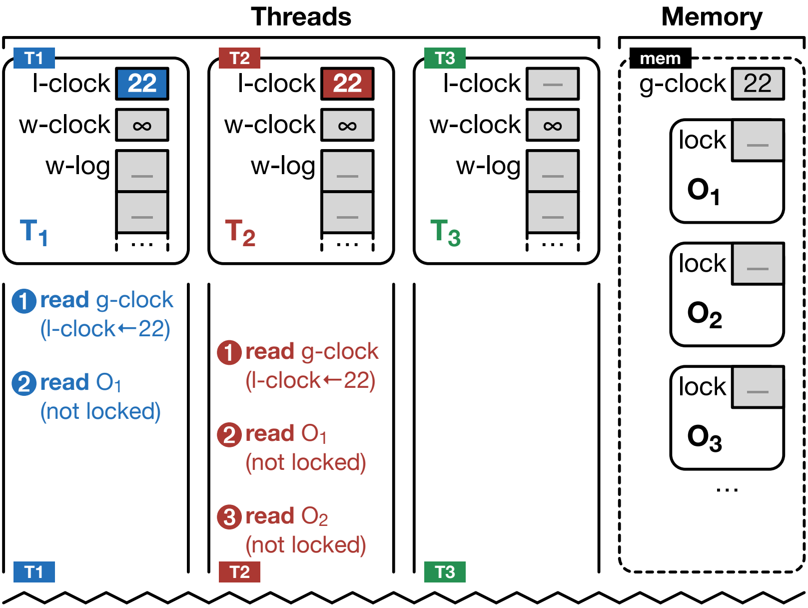
```
/* ... some important code that
   we consider later ... */

/* Update references */
rlu_assign_ptr(&(new->next), next);
rlu_assign_ptr(&(prev->next), new);
/* Commit */
rlu_reader_unlock();
```

Table of Contents

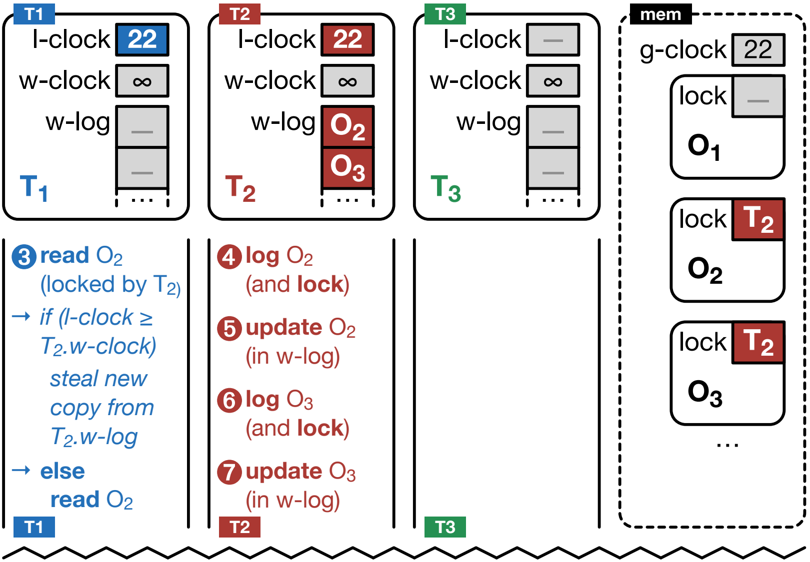
Basic idea

1. All operations read the global clock when they start;
2. Clock is used to dereference shared objects;
3. Write operations write to a log (RCU-style copy of an object);
4. Increment global clock to commit write (Swap pointers in RCU);
5. Wait old readers to finish (`synchronize_rcu`);
6. Write-back objects from the log. (Corresponds to RCU memory reclamation)

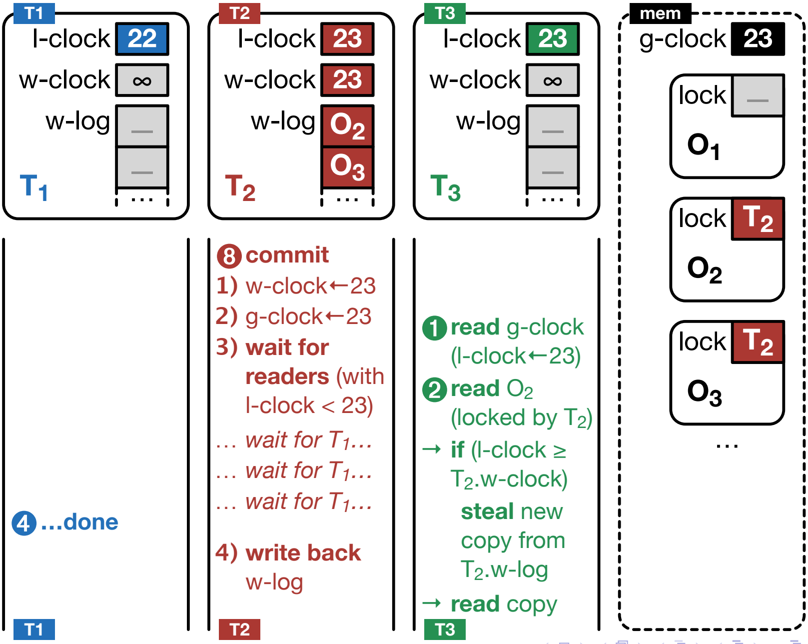
Read-read example



Write-read example



Read-write-steal example



Real list add

```
int rlu_list_add(rlu_thread_data_t *self,
                 list_t *list, val_t val) {
    node_t *prev, *next, *node;
    val_t v;
restart:
    rlu_reader_lock();
    /* Find right place ... */
    if (!rlu_try_lock(self, &prev) ||
        !rlu_try_lock(self, &next)) {
        rlu_abort(self);
        goto restart;
    }
    new = rlu_new_node(); new->val = val;
    rlu_assign_ptr(&(new->next), next);
    rlu_assign_ptr(&(prev->next), new);
    rlu_reader_unlock();
}
```

Real list add

```
int rlu_list_add(rlu_thread_data_t *self,
                 list_t *list, val_t val) {
    node_t *prev, *next, *node;
    val_t v;
restart:
    rlu_reader_lock();
    /* Find right place... */
    if (!rlu_try_lock(self, &prev) ||
        !rlu_try_lock(self, &next)) {
        rlu_abort(self);
        goto restart;
    }
    new = rlu_new_node(); new->val = val;
    rlu_assign_ptr(&(new->next), next);
    rlu_assign_ptr(&(prev->next), new);
    rlu_reader_unlock();
}
```

Real list add

```
int rlu_list_add(rlu_thread_data_t *self,
                 list_t *list, val_t val) {
    node_t *prev, *next, *node;
    val_t v;
restart:
    rlu_reader_lock();
    /* Find right place... */
    if (!rlu_try_lock(self, &prev) ||
        !rlu_try_lock(self, &next)) {
        rlu_abort(self);
        goto restart;
    }
    new = rlu_new_node(); new->val = val;
    rlu_assign_ptr(&(new->next), next);
    rlu_assign_ptr(&(prev->next), new);
    rlu_reader_unlock();
}
```

Reader lock

```
function RLU_READER_LOCK(ctx)
    ctx.is-writer ← false
    ctx.run-cnt ← ctx.run-cnt +1                          ▷ Set active
    memory fence
    ctx.local-clock ← global-clock              ▷ Record global clock
function RLU_READER_UNLOCK(ctx)
    ctx.run-cnt ← ctx.run-cnt +1                        ▷ Set inactive
    if ctx.is-writer then
        RLU_COMMIT_WRITE_LOG(ctx)                     ▷ Write updates
```

Memory commit

```
function RLU_COMMIT_WRITE_LOG(ctx)
    ctx.write-clock ← global-clock +1          ▷ Enable stealing
    FETCH_AND_ADD(global-clock, 1)             ▷ Advance clock
    RLU_SYNCHRONIZE(ctx)                       ▷ Drain readers
    RLU_WRITEBACK_WRITE_LOG(ctx)               ▷ Safe to write back
    RLU_UNLOCK_WRITE_LOG(ctx)
    ctx.write-clock ← ∞                        ▷ Disable stealing
    RLU_SWAP_WRITE_LOGS(ctx)                   ▷ Quiesce write-log
```

Pointer dereference

```
function RLU_DEREFERENCE(ctx, obj)
    ptr-copy ← GET_COPY(obj)                          ▷ Get copy pointer
    if IS_UNLOCKED(ptr-copy) then                     ▷ Is free?
        return obj                                    ▷ Yes ⇒ return object
    if IS_COPY(ptr-copy) then                         ▷ Already a copy?
        return obj                                    ▷ Yes ⇒ return object
    thr-id ← GET_THREAD_ID(ptr-copy)
    if thr-id = ctx.thr-id then                       ▷ Locked by us?
        return ptr-copy                               ▷ Yes ⇒ return copy
    other-ctx ← GET_CTX(thr-id)                       ▷ No ⇒ check for steal
    if other-ctx.write-clock ≤ ctx.local-clock then
        return ptr-copy                               ▷ Stealing ⇒ return copy
    return obj                                        ▷ No stealing ⇒ return object
```

RLU Deferring

1. On commit do not increment the global clock and execute RLU sync;
2. Instead, save writer-log and create a new log for the next writer
3. Synchronize when a writer tries to lock an object that is already locked.

RLU Deferring advantages

1. Reduce the amount of RLU synchronize calls
2. Reduce contention on a global clock
3. Less stealing – less cache misses

Table of Contents

Linked lists

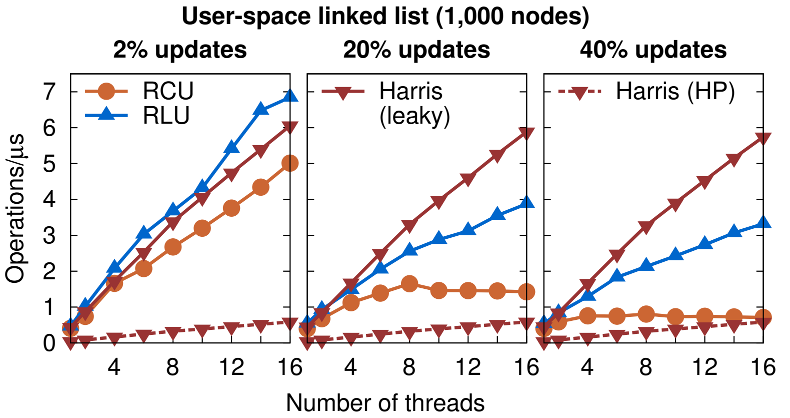


Figure 4. Throughput for linked lists with 2% (left), 20% (middle), and 40% (right) updates.

Hash table

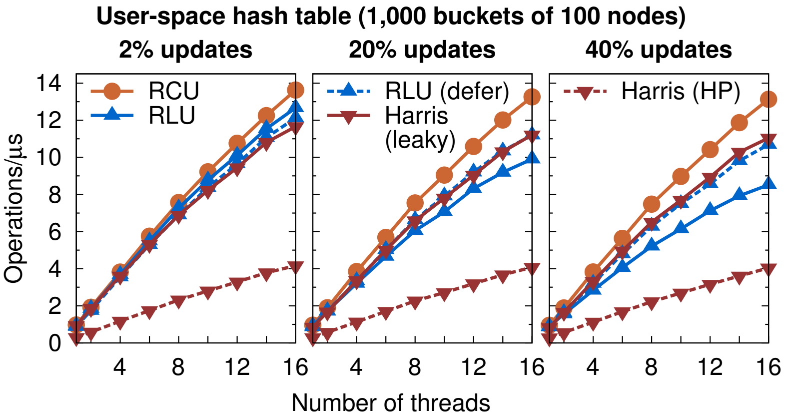


Figure 5. Throughput for hash tables with 2% (left), 20% (middle), and 40% (right) updates.

Resizable Hash table

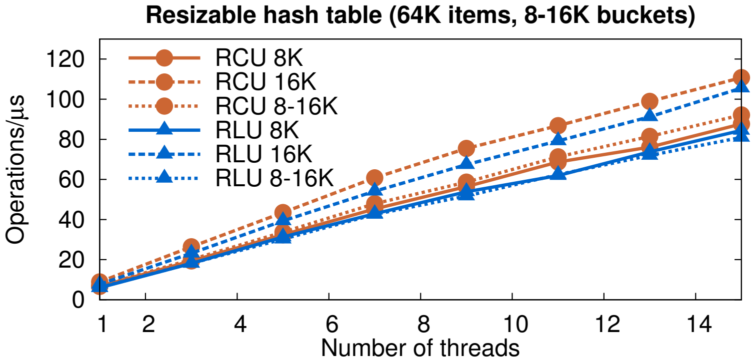


Figure 6. Throughput for the resizable hash table.

Update only stress test (hash table)

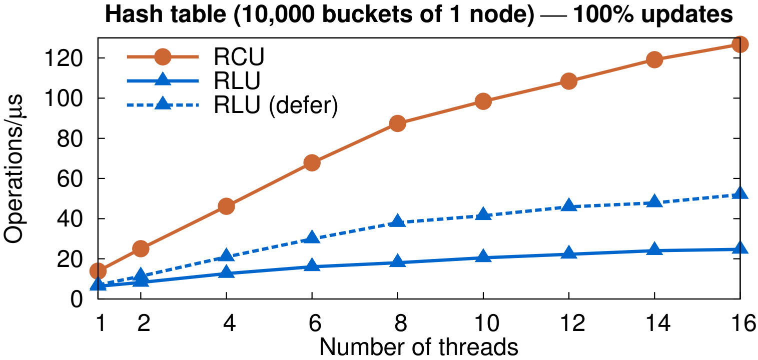
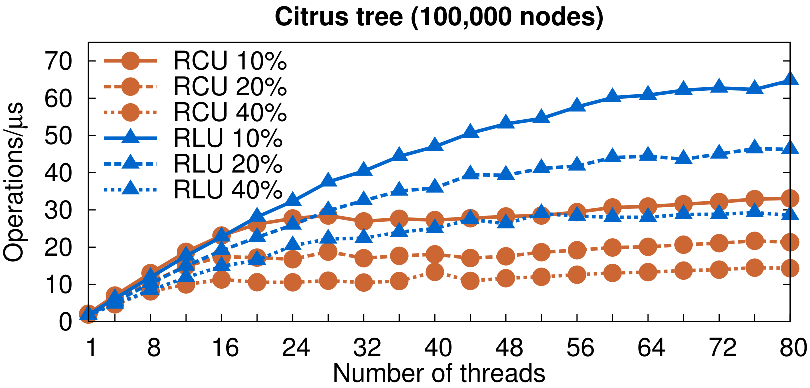
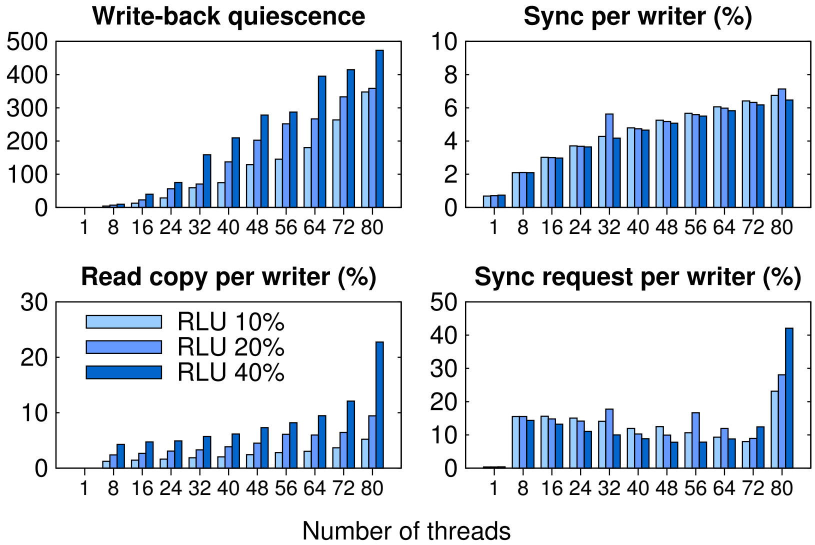


Figure 7. Throughput for the stress test on a hash table with 100% updates and a single item per bucket.

Citrus Search Tree (throughput)



Citrus Search Tree (statistics)



Kernel space doubly linked lists

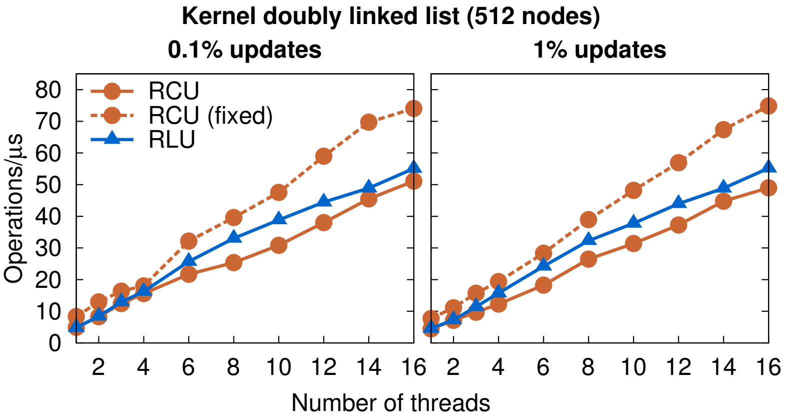


Figure 9. Throughput for kernel doubly linked lists (`list_*` APIs) with 0.1% (left) and 1% (right) updates.

Kernel space single linked lists

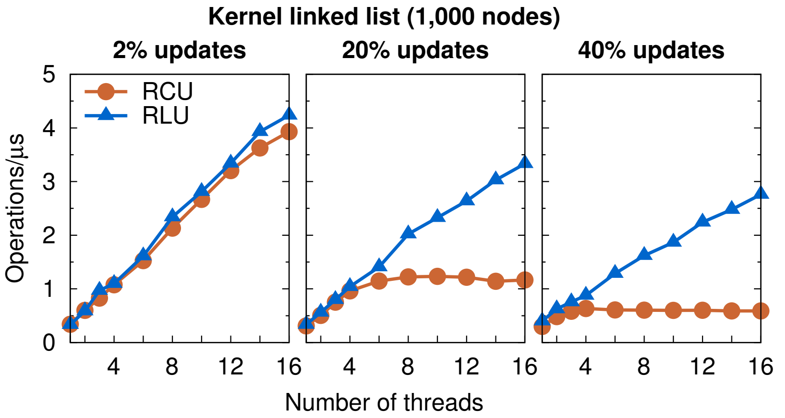


Figure 10. Throughput for linked lists running in the kernel with 2% (left), 20% (middle), and 40% (right) updates.

Kernel space hash tables

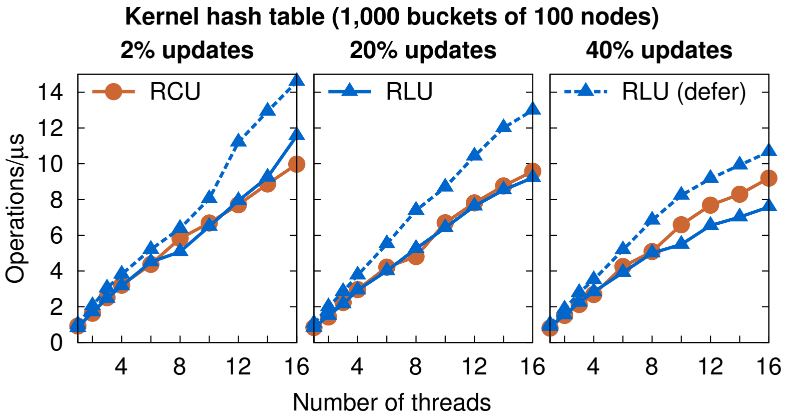


Figure 11. Throughput for hash tables running in the kernel with 2% (left), 20% (middle), and 40% (right) updates.

Kernel-space Torture Tests

They tried even this!
RLU successfully passed all of the within implemented functionality.

Kyoto Cache DB

It was advertised in the abstract and finally here it is:

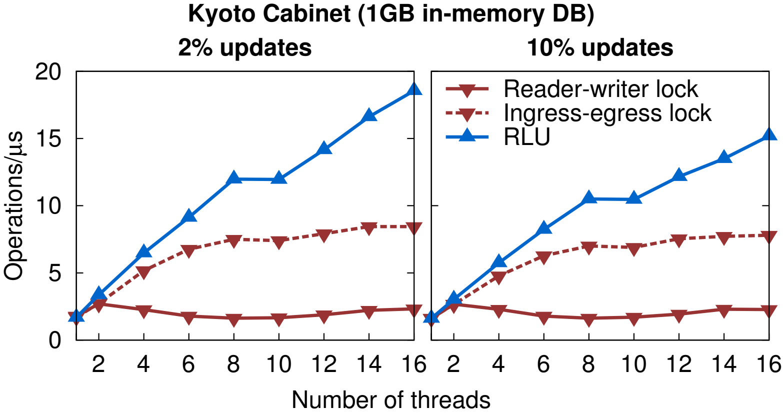


Figure 12. Throughput for the original and RLU versions of the Kyoto Cache DB.

Table of Contents

Conclusion

- Performance similar to RCU.
  Sometimes better, sometimes worse.
- Easier programming interface
- Compatible with RCU
- Good both in user and kernel space
- Severely benchmarked.